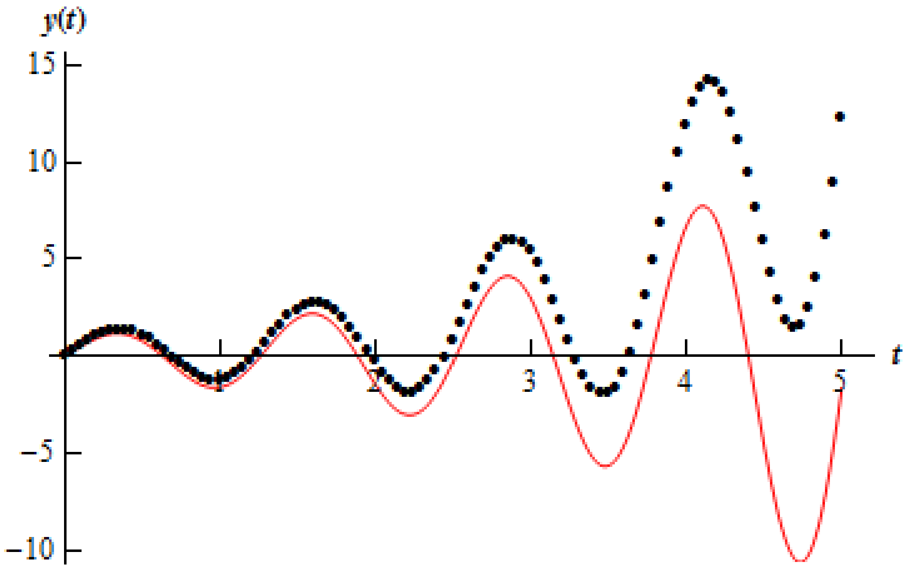
Physics based animation
Unit 03 - Time and integration
Practical

Grégory Leplâtre

Semester 1 - 2017/2018



1 Introduction

The Goals of this first practical are to:

- Modify the architecture to suit our evolving requirements.
- Implement a robust approach to timing.
- Develop a practical understanding of common numerical integration methods

2 Tasks

2.1 Task 1 - Architecture

Some of the operations that we will need to complete today and in the near future will stretch the functionalities of our architecture. In fact, you might have faced issues already. For example, if you try to simulate a collection of particles, you need to keep track of the particles' positions, which a Mesh object allows you to do; but you also need to keep track of velocities, mass, etc, which mesh objects have no concept of.

There are two options:

1. Add everything we need to the Mesh class
2. Create a Particle class that represents the objects we are dealing with here.

The first option might seem like the easiest one as it doesn't require us to create a new class, but it is not the most elegant one. Conceptually, a Mesh object is supposed to represent a 3D object located in space, destined to be rendered. A mesh doesn't weigh anything, so applying forces to one doesn't really make sense.

What we need is a concept of *particle*, which is an object that has a position, but also a mass and to which one can apply forces that will affect its acceleration, velocity and position. Before we implement this class, let's consider another type of object that we will work with later in the module to save us a bit of refactoring later on: rigid bodies. So the two types of entities we will be working with are:

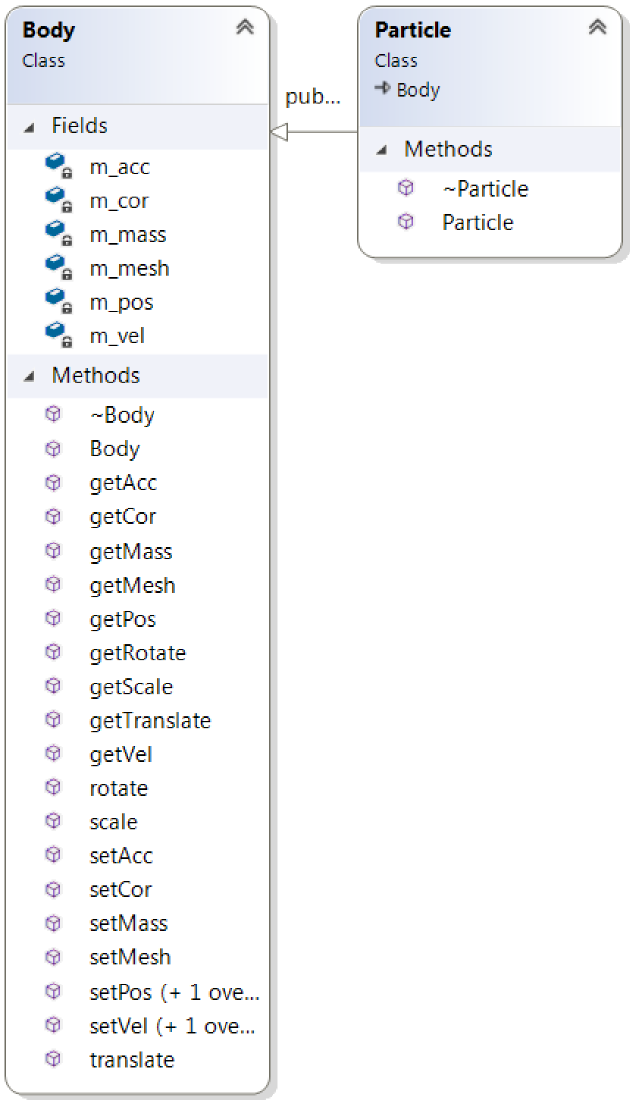
1. *Particles*: dimensionless[1] objects that have a mass, acceleration, velocity and position.
2. *Rigid Bodies*: objects that have a dimension, mass, acceleration, velocity, position ... and more. These objects typically represent physical objects that we are used to in the real-world. The fact that they have a dimension means that we will need to store additional properties in order to simulate their physical behaviour. For example, their density and inertia tensor. You will see in a few weeks...

As you can see, these objects share properties, so it makes sense for them to inherit a common base class. To make sure we all share a similar architecture going forward, here's the class diagram that I would like you to implement.

Task: Implement a `Body` and `Particle` class according to the class diagram presented below.

[1]Although they are conceptually dimensionless, we have represented them with a non-dimensionless mesh so far ... Otherwise we wouldn't be able to see them!

The outcome of the task should be a simulation that does exactly the same as the one you produced last week (a particle bouncing around inside a cube), but with an improved architecture. This is actually a trivial task as I'm providing you with the code for both classes, to ensure that we are on the same page. All you have to do is update your main file so that you are using `Particle` objects and not `Mesh` objects in your simulation. The added benefit is that, since `Particle` objects keep track of acceleration and velocity, you won't have to create separate data structures for these.



The `Body` class header file is provided to you for convenience, and again, to ensure that your architecture evolves consistently with the one that will be presented in subsequent weeks. As you can see, I tend to implement the functions that fit in one line in the header file, which means there is little you need to implement in `Body.cpp`.

```cpp
  /*
  ** GET METHODS
  */
  // mesh
  Mesh& getMesh() { return m_mesh; }

  // transform matrices
  glm::mat3 getTranslate() const { return m_mesh.getTranslate(); }
  glm::mat3 getRotate() const { return m_mesh.getRotate(); }
  glm::mat3 getScale() const { return m_mesh.getScale(); }

  // dynamic variables
  glm::vec3& getAcc() { return m_acc; }
  glm::vec3& getVel() { return m_vel; }
  glm::vec3& getPos() { return m_pos; }


  // physical properties
  float getMass() const { return m_mass; }
  float getCor() { return m_cor; }

  /*
  ** SET METHODS
  */
  // mesh
  void setMesh(Mesh m) { m_mesh = m; }

  // dynamic variables
  void setAcc(const glm::vec3 &vect) { m_acc = vect; }
  void setVel(const glm::vec3 &vect) { m_vel = vect; }
  void setVel(int i, float v) { m_vel[i] = v; } //set the ith coordinate of the velocity vector
  void setPos(const glm::vec3 &vect) { m_pos = vect; m_mesh.setPos(vect); }
  void setPos(int i, float p) { m_pos[i] = p; m_mesh.setPos(i, p); } //set the ith coordinate of the position vector

  // physical properties
  void setCor(float cor) { m_cor = cor; }
  void setMass(float mass) { m_mass = mass; }

  /*
  ** OTHER METHODS
  */

  // transformation methods
  void translate(const glm::vec3 &vect);
  void rotate(float angle, const glm::vec3 &vect);
  void scale(const glm::vec3 &vect);

private:
  Mesh m_mesh; // mesh used to represent the body

  float m_mass; // mass
  float m_cor; // coefficient of restitution

  glm::vec3 m_acc; // acceleration
  glm::vec3 m_vel; // velocity
  glm::vec3 m_pos; // position
};
```

Here's `Body.cpp`:

```cpp
#include "Body.h"

Body::Body()
{
}

Body::~Body()
{
}

/* TRANSFORMATION METHODS*/
void Body::translate(const glm::vec3 &vect) {
  m_pos = m_pos + vect;
  m_mesh.translate(vect);
}

void Body::rotate(float angle, const glm::vec3 &vect) {
  m_mesh.rotate(angle, vect);
}

void Body::scale(const glm::vec3 &vect) {
  m_mesh.scale(vect);
}
```

The `Particle` class is trivial as at the moment we don't have any reason to add any information to that class that doesn't already belong to `Body`. Here's `Particle.h`

```cpp
#pragma once
#include "Body.h"
class Particle :
  public Body
{
public:
  Particle();
  ~Particle();
};
```

And here's `Particle.cpp`, which only contains the default `Particle` constructor.

```cpp
// Math constants
#define _USE_MATH_DEFINES
#include <cmath>
#include "Particle.h"


// default constructor: creates a particle represented by a default (square).
// Notes:
// - particle rotated so that it is orthogonal to the z axis.
// - scaled
// - no shader allocated by default to avoid creating a Shader object for each particle.
Particle::Particle()
{
  setMesh(Mesh::Mesh());
  scale(glm::vec3(.1f, .1f, .1f));
  rotate((GLfloat)M_PI_2, glm::vec3(1.0f, 0.0f, 0.0f));

  // set dynamic values
  setAcc(glm::vec3(0.0f, 0.0f, 0.0f));
  setVel(glm::vec3(0.0f, 0.0f, 0.0f));

  // physical properties
  setMass(1.0f);
  setCor(1.0f);
}

Particle::~Particle()
{
}
```

2.2 Task 2 - Time step

The performance of integration techniques is correlated to the size of the time step used. The smaller the timestep, the better integrators behave. The problem is that the timestep is not an independent variable of our simulation. It depends on the frequency at which we can perform our game loop, which depends on everything that goes on within the game loop, most notably physics computation and rendering. In our case, rendering will not be the problem, but physics computations will.

The following page provides a good presentation of various approaches and their limitations: `https://gafferongames.com/post/fix_your_timestep/`. Please read the page in order to understand the problem and potential solutions.

Task: Implement the time step solution presented in the "Free The Physics" section of the above page.

You are welcome to implement the final solution presented in that article "The final touch", but you'll have to implement a concept of *State*, which you are not expected to do at this stage.

2.3 Task 3 - Integration methods

As we have seen in the lecture, different integration methods are available and their performance is variable depending on the time step, but also on the type of simulation implemented. The purpose of this task is to give you an opportunity to witness the differences between simple integration techniques. You will do this by running the same simple simulation with two integrators: **Forward Euler** and **Semi-implicit Euler**.

Task: Implement two bouncing particle simulations, the only difference between each simulation being the integration method used.

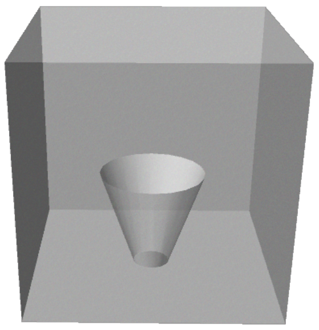
Here's information about how you should proceed:

1. create 3 particles separated by a few pixels on the x axis. In other words, the particles will be next to each other on your screen.
2. Translate each particle so that it sits 2 units above the ground plane.
3. Keep the first one immobile (don't apply any force to it), it will act as a reference.
4. Apply gravity to the other 2 and use collision detection with the ground plane to ensure that the particles bounce on the plane without any loss of momentum. Give the 3 particles an initial velocity of (0,0,0).
5. Run your simulation for various timestep values, say from 0.0005 to 0.1 and observe the differences of behaviours.

2.4 Task 4 - Blow dryer

The final task will demand a little bit of creativity to turn a high-level description of the system's requirements into a working application. A particle bouncing around in a cubic room is not that exciting. What if one carved a hole in the middle of the floor, stuck a blow dryer through it and switched it on. This is what I'm asking you to do. The conical shape depicted on the Figure below represent the limits of the area in which the blow dryer has an effect. You can make the following assumptions about its behaviour:

- The force it generates is maximal at the bottom of the cone and null at the top. The rate of decrease of the force field along the vertical axis is up to your experimentation.
- The force is maximal along the central vertical axis of the cone at any given height and decreases radially until it is null at the edge of the cone.
- You can assume that the force applied is always vertical (and directed upwards) at any position within the cone. An extra mark (and therefore the possibility of scoring 16 out of 15 is available for anyone who will model a force whose direction is vertical on the central axis and becomes parallel to the cone edge at the edges.



Task: Simulate a "blow dryer" effect as described above and add it to the room.

You'll need to experiment with the strength of the blow, the size of its area of influence until you achieve an interesting behaviour when a particle bounces around within the cube.

3 Deliverables

This week's work is assessed, so you need to submit evidence of completed tasks to gain credits.

3.1 Marking scheme

here's a summary of what you need to deliver and allocated marks:

- **Particle bouncing within cube - 6 marks**
  You will get these marks by simply showing the outcome of the previous lab to me in the lab
- **Implementation of prescribed architecture (Tasks 1 and 2) - 3 marks**
  Show me your main, build it and run it and if the result is still a bouncing particle, you will get 3 marks.
- **implementation of 2 integration methods (Task 3) - 3 marks** Marks granted on demonstration of the outcome of Task 3.
- **Addition of a blow-dryer effect (Task 4) - 3 marks** Marks granted on demonstration of the outcome of Task 4.

3.2 Submission details

You must submit your work using the relevant Moodle assignment by the deadline specified on Moodle (Sunday the 1st of October at 5pm). Please submit your zipped Visual Studio project. It is up to you how you organise your submission. You may submit several projects, or a single one with different mains, or organise your main to that it calls a different function for each task. The point of the Moodle submission is for me to verify that you are the author of the code that you will run when you demonstrate your working code to me in the lab. **Marks will only be granted when a working application is demonstrated in the lab**.